APPENDIX

```
/*
 *
 *        Copyright (c) NEC America, Inc 2000
 *            All Rights Reserved
 *
 * No part of this program may be photocopied, reproduced, or
 * translated to another programming language without the prior
 * written consent of NEC America, Inc.
 *
 * FILE:                compress.c
 *
 * CONTENTS:
 *
 *    Implementation of compression / decompression API.
 *
 *
 * CONSTRAINTS:
 *
 *
 * RESTRICTIONS/PROBLEMS:
 *
 *
 * RELATED FILES:
 *
 *
 * REVISION HISTORY:
 *
 *
 *
 */

/*=============================================================================
=============
       1       2       3       4       5       6       7       8
12345678901234567890123456789012345678901234567890123456789012345678901234567890123
4567890
=============================================================================
=============*/

```

```
/*****************************************************************************
***
 * Includes
 */
#include <stdlib.h>
#include <string.h>
#include <bstring.h>
#include "global.h"
#include "xoprintf.h"
#include "basictypes.h"

/*****************************************************************************
***
 * Defines
 */

#define MAX_BITS_PER_CODE 14        /* The size of each code, 9 to 15 bits */

/* total number of codes in the dictionary */
#define MAX_CODES    (1 << MAX_BITS_PER_CODE)

/* this code is reserved for 'null code', also used as 'EOF code' */
#define NULL_CODE    (256)

/* number of predefine codes representing 2 .. 64 zeros
 * these codes are codes 257 .. (257+NUM_ZERO_CODES-1)
 */
#define NUM_ZERO_CODES (63)

/* the first code to be dynamically defined in run time */
#define FIRST_USER_CODE (256 + 1 + NUM_ZERO_CODES)

/* the last user code to be dynamically defined in run time */
#define LAST_CODE    (MAX_CODES - 1)
```

```
/*****************************************************************************
***
 * Typedefs
 */

// A code is used to represent a string of bytes.  If a code (e.g., the
// first 256 codes) represents only one byte, its prefixCode will be
// NULL_CODE, indicating
// that there is no prefixCode.  For example, if code x represents 'T',
// code y represents 'TH', and code z represents 'THE', the codeDef of these
// codes would have been:
// x:  prefixCode[x] = NULL_CODE
//     appendCode[x] = 'T'
// y:  prefixCode[y] = x
//     appendCode[y] = 'H'
// z:  prefixCode[z] = y
//     appendCode[z] = 'E'
// The EOF code will be NULL_CODE, which requires special processing
// NULL_CODE: prefixCode = NULL_CODE
//            appendCode = NULL_CODE
//
// For LZWK, we can append any code with another code.  (e.g., if
// code x = "XYZ", code for "XYZXYZ" can be
// coded as prefixCode = x, appendCode = x.)  This way, we can represent
// repeating patterns better.
//
//
// The definitions of all possible codes are stored in the following arrays.
// The definitions of the codes are sometimes referred to as the 'dictionary'
// The first 256 codes is reserved for the 256 possible values in a byte.
// because of the nature of the MIB, long strings of 0's is likely to occur,
// we also pre-define these codes in the dictionary:
// code (NULL_CODE) is reserved for NULL code and EOF code
// code 257: represents 2   bytes of zeros
// code 258: represents 3   bytes of zeros
// code 259: represents 5   bytes of zeros
// code 260: represents 6   bytes of zeros
// code 261: represents 7   bytes of zeros
// code 262: represents 8   bytes of zeros
// code 263: represents 9   bytes of zeros
// code 264: represents 10  bytes of zeros
// code 265: represents 11  bytes of zeros
// code 266: represents 12  bytes of zeros
// code 267: represents 13  bytes of zeros
```

```
// .
// .
// .
// code FIRST_USER_CODE) through (LAST_CODE) is dynamically assigned
//
// The following type is used to define a table which stores the
// definitions of all codes.
// myCode is not needed for compression.  It's just for debugging.
//

typedef struct {
   U16T   myCode;     // for easier debugging
   U16T   prefixCode;  // what code is this code is built on (parent code)
   U16T   appendCode;  // the additional string on top of prefixCode
   U16T   siblingCode; // other codes that have the same parent code
   U32T   codeSize;   // length of the string represented by this code
   U8T   * codeStrP;  // address of a string represented by this code
   U16T   childCode[16]; // codes that build on this code
} CodeDefT;
```

```
/*****************************************************************************
***
 * Static Data
 */

// The dictionary definition.  Stores the definition of each code.
CodeDefT CodeDefTbl[MAX_CODES];

// Some constant tables
const U8T zeroBytes[NUM_ZERO_CODES+1] = {0};

const U8T byteVals[256] = {
        0x00,  0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08,  0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
        0x10,  0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
        0x18,  0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
        0x20,  0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
        0x28,  0x29, 0x2A, 0x2B, 0x2C, 0x2D, 0x2E, 0x2F,
        0x30,  0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
        0x38,  0x39, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F,
        0x40,  0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
        0x48,  0x49, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F,
        0x50,  0x51, 0x52, 0x53, 0x54, 0x55, 0x56, 0x57,
        0x58,  0x59, 0x5A, 0x5B, 0x5C, 0x5D, 0x5E, 0x5F,
        0x60,  0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
        0x68,  0x69, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F,
        0x70,  0x71, 0x72, 0x73, 0x74, 0x75, 0x76, 0x77,
        0x78,  0x79, 0x7A, 0x7B, 0x7C, 0x7D, 0x7E, 0x7F,
        0x80,  0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x87,
        0x88,  0x89, 0x8A, 0x8B, 0x8C, 0x8D, 0x8E, 0x8F,
        0x90,  0x91, 0x92, 0x93, 0x94, 0x95, 0x96, 0x97,
        0x98,  0x99, 0x9A, 0x9B, 0x9C, 0x9D, 0x9E, 0x9F,
        0xA0,  0xA1, 0xA2, 0xA3, 0xA4, 0xA5, 0xA6, 0xA7,
        0xA8,  0xA9, 0xAA, 0xAB, 0xAC, 0xAD, 0xAE, 0xAF,
        0xB0,  0xB1, 0xB2, 0xB3, 0xB4, 0xB5, 0xB6, 0xB7,
        0xB8,  0xB9, 0xBA, 0xBB, 0xBC, 0xBD, 0xBE, 0xBF,
        0xC0,  0xC1, 0xC2, 0xC3, 0xC4, 0xC5, 0xC6, 0xC7,
        0xC8,  0xC9, 0xCA, 0xCB, 0xCC, 0xCD, 0xCE, 0xCF,
        0xD0,  0xD1, 0xD2, 0xD3, 0xD4, 0xD5, 0xD6, 0xD7,
        0xD8,  0xD9, 0xDA, 0xDB, 0xDC, 0xDD, 0xDE, 0xDF,
        0xE0,  0xE1, 0xE2, 0xE3, 0xE4, 0xE5, 0xE6, 0xE7,
        0xE8,  0xE9, 0xEA, 0xEB, 0xEC, 0xED, 0xEE, 0xEF,
        0xF0,  0xF1, 0xF2, 0xF3, 0xF4, 0xF5, 0xF6, 0xF7,
        0xF8,  0xF9, 0xFA, 0xFB, 0xFC, 0xFD, 0xFE, 0xFF
    };
```

```
/*
*=============================================================================
=========
 * Function Name:      Send
 *
 * Description:
 *      Send up to 32 bits of data into a memory string.
 *      sending from least significant bit before MSb
 *      starting from lower address byte toward higher address byte
 *
 * Inputs:
 *      fBits: the data
 *      fNumBits: how many bits to send
 *      fBytePP: pointer to the pointer of where to store the data
 *      fNextBitPos: pointer to the data for bit position of where to store
 *
 * Outputs:
 *      none
 *
 * Callable From:      any task
 *
 * Reentrant:   Yes
 *
*=============================================================================
=========
 */
PRIVATE void Send (
   U32T fBits, // up to 32 bits of data, transmitted from LSb to MSb
   U32T fNumBits, // 1 .. 32
   U8T ** fBytePP,
   U32T * fNextBitPos // 0 => LSb, 7 => MSb
   )
{
   register U32T BitsVal /* = fBits */;
   register U32T NumBits /* = fNumBits */;
   register int  BitsSent;
   register U8T  ByteVal;
   register int  NextBitPos;

   BitsVal = fBits;
   NumBits = fNumBits;
   NextBitPos = (*fNextBitPos);
```

```
    while (NumBits) {
        // decide how many bits to send this time
        BitsSent = 8 - NextBitPos;
        if (NumBits <= BitsSent)
          BitsSent = NumBits;

        // extracts and copy the bits to be sent this time
        ByteVal = (U8T) (BitsVal & ((1 << BitsSent) - 1));
        if (NextBitPos == 0)
          (**fBytePP) = ByteVal;
        else
          (**fBytePP) += ByteVal << NextBitPos;

        // adjust BitsVal, NumBits, NextBitPos, and *fBytePP
        BitsVal = BitsVal >> BitsSent;
        NumBits -= BitsSent;
        NextBitPos += BitsSent;
        if (NextBitPos >= 8) {
          NextBitPos = 0;
          (*fBytePP)++;
        }
    }
    (*fNextBitPos) = NextBitPos;
}
```

```
/*
*=============================================================================
=========
 * Function Name:      Receive
 *
 * Description:
 *       extract up to 32 bits of data from a a memory string.
 *       sending from least significant bit before MSb
 *       starting from lower address byte toward higher address byte
 *
 * Inputs:
 *       fNumBits: how many bits to receive
 *       fBytePP: pointer to the pointer of where to read the data
 *       fNextBitPos: pointer to the data for bit position of where to read
 *
 * Outputs:
 *       the data extracted
 *
 * Callable From:      any task
 *
 * Reentrant:   Yes
 *

*=============================================================================
=========
 */
PRIVATE U32T Receive ( // returns up to 32 bits of data,
   U32T fNumBits, // 1 .. 32
   U8T ** fBytePP,
   U32T * fNextBitPos // 0 => LSb, 7 => MSb
   )
{
   register U32T BitsVal;
   register int  ShiftBits;
   register U8T  *ByteP = *fBytePP;

   // read the first 16 bits
   ShiftBits = (*fNextBitPos);
   BitsVal = ( ( ( (U32T) (*ByteP++) ) ) +
               ( ( (U32T) (*ByteP++) ) << 8) ) >> ShiftBits;
   ShiftBits = 16 - ShiftBits; // The number of bits we've got so far

   while (fNumBits > ShiftBits) {
```

```
        BitsVal   += ((U32T) (*ByteP++)) << ShiftBits;
        ShiftBits += 8;
    }

    BitsVal = BitsVal & ((1 << fNumBits) - 1);

    // adjust *fBytePP and *fNextBitPos
    (*fNextBitPos) += (fNumBits & 7);
    (*fBytePP) += (fNumBits >> 3) + ((*fNextBitPos) >> 3);
    (*fNextBitPos) = (*fNextBitPos) & 7;

    return BitsVal;

}
```

```
/*
*=============================================================================
=============
 * Macro Name:        HASH_OF
 *
 * Description:
 *       Calculate a 4-bit hash index based on a given byte value
 *
 * Inputs:
 *       fChar, a byte value
 *
 * Outputs:
 *       a 4-bit value
 *
 * Callable From:
 *            anywhere
 *
 * Reentrant:   yes
 *
*=============================================================================
=============
 */

#define HASH_OF(fChar) (((fChar) ^ ((fChar) >> 4)) & 0x0000000F)
```

```
/*
*=============================================================================
 =============
 * Function Name:      FindBestFamilyMember
 *
 * Description:
 *       When a string code (fParentCode) matches the input string,
 *    this routine is called to
 *       find the best (longest) match code in the input dictionary
 *    that matches the input string.
 *       This routine searches all the children of fParentCode to see if
 *    any one of them matches better than fParentCode.  If so, the best
 *    matched code and the string size of the code is returned.
 *    Otherwise, fParentCode and its string size is returned.
 *
 *       Note:
 *       The parent code must not be NULL_CODE
 *       The parent code should already match the input string prior
 *    to the entry of this routine
 *       The fInputStrP is the address of the string, including parent string
 *    not just the append string of the children
 *       The way code definitions are stored allows us to make the assumption
 *       that the first child that matches the input string
 *       will be the best-match child among all the children of the same parent.
 *    (see notes at the beginning of this file)
 *
 * Inputs:
 *   U16T fParentCode,  the code that we have found matched
 *   U8T  * fInputStrP, pointer to the whole string in the input
 *   int  fBytesLeft,   bytes left in the whole input
 *
 * Outputs:      the best-match code
 *
 * Callable From:       anywhere
 *
 * Reentrant:   yes, recursion is OK
 *
*=============================================================================
 =============
 */
U16T FindBestFamilyMember (
   U16T fParentCode, /* the code that we have found matched */
```

```
    U8T  * fInputStrP, /* pointer to the whole string */
    int  fBytesLeft  /* bytes left in the whole input */
    )
{
    register U16T myBestCode;
    register int  myBestSize;
    register U16T myChild;
    register int  myChildSize;
    register U16T childAppendCode;

    // the parent code has already matched the input string
    // now let's see if any of the children matches better
    //
    // Get the first characters of the remaining string
    // find the hash index of the character
    // use that hash index to find the first child code of fParentCode
    // if myChild is NULL_CODE, fParentCode is the best match
    // Otherwise, we need do more code-string comparison to find out
    // if any child is the best match
    //
    // get the first child
    myBestCode =  fParentCode;
    myBestSize =  CodeDefTbl[myBestCode].codeSize;
    myChild =CodeDefTbl[myBestCode].childCode[HASH_OF(fInputStrP[myBestSize])];

    // search children and children of children until the best-match is found
    while ( (myChild != NULL_CODE) && (fBytesLeft > myBestSize) ) {

        myChildSize =  CodeDefTbl[myChild].codeSize;
        childAppendCode =  CodeDefTbl[myChild].appendCode;

        // if this child of myBestCode matches input string
        // we have found a better than myBestCode
        if ( (myChildSize <= fBytesLeft) &&
            (memcmp(&(fInputStrP[myBestSize]),
                    CodeDefTbl[childAppendCode].codeStrP,
                    CodeDefTbl[childAppendCode].codeSize) == 0) ) {
            // the first-match child is where we can find the best-match child
            // this is based on the assumption that new children are always
            // added to the end of the children list in addCodeLZWK
            // see notes in addCodeLZWK for explaination
            myBestCode = myChild;
            myBestSize = myChildSize;
            myChild    = CodeDefTbl[myBestCode].childCode[
                HASH_OF(fInputStrP[myBestSize]) ];
```

```
            } else {
              // else, this child does not match, look at the next child
              myChild = CodeDefTbl[myChild].siblingCode;
            }
        }

        return (myBestCode);
    }
```

```
/*
*=============================================================================
 ============
 * Function Name:     addCodeLZWK
 *
 * Description:
 *       add a new code to the dictionary
 *
 *       Note:
 *       When adding a child to a family, we always add the new child as the
 *       last child.  This will assure us if any child (child A) represent
 *       a string that is a superset of the string represented by
 *       another child (child B), child A will be visited before child B can
 *       be visited.
 *       With this assumption in mind, our string matching routine do
 *       not have to recursively search the entire family to find the best
 *       match.  We only need to find the first-match child, knowing either this
 *       child, or one of its descendants will be the best-match.
 *
 * Inputs:
 *
 * Outputs:
 *
 * Callable From:
 *
 * Reentrant:
 *

*=============================================================================
 ============
 */
PRIVATE void addCodeLZWK(
   U16T fnewCode,
   U8T  * fStrP,     /* the address of the string represented by this code */
   U32T fCodeSize,   /* number of bytes of this string */
   U16T fPrefixCode, /* the parent of this code */
   U16T fAppendCode  /* the append code of this code */
)
{
   register int index;
   register CodeDefT * myCodeDefP = &(CodeDefTbl[fnewCode]);
```

```
    // initialize the structure of the new code
    myCodeDefP->prefixCode  = fPrefixCode;
    myCodeDefP->appendCode  = fAppendCode;
    myCodeDefP->codeStrP    = fStrP;
    myCodeDefP->codeSize    = fCodeSize;
    // myCodeDefP->siblingCode = NULL_CODE;       already initialized
//  for (index = 0; index < 16; index++)        already initialized
//      myCodeDefP->childCode[index] = NULL_CODE;

    // insert the new code to be a child of the parent.
    // the hash index is determined by the first byte of the append code
    // Note: It is important to add this child as the last child
    // FindBestFamilyMember is depending on that assumption
    if (fPrefixCode != NULL_CODE) {
        register CodeDefT * parentCodeDefP = &(CodeDefTbl[fPrefixCode]);
        register U16T lastChild;
        U16T tempChild;

        index = HASH_OF(fStrP[parentCodeDefP->codeSize]);
        lastChild = parentCodeDefP->childCode[index];
        if (lastChild == NULL_CODE) { // parent has no child yet
          parentCodeDefP->childCode[index] = fnewCode;
        } else { // parent has other child
          // find the last child in the list
          while ( (tempChild = CodeDefTbl[lastChild].siblingCode) !=
                    NULL_CODE) {
              lastChild = tempChild;
          }
          // add the new child after lastChild as the new last child
          CodeDefTbl[lastChild].siblingCode = fnewCode;
        }
        myCodeDefP->siblingCode = NULL_CODE;
    }
}
```

```
/*
*=============================================================================
 ========
 * Function Name:     initLZWK
 *
 * Description:
 *       Init process to be done before doing compression/decompression
 *
 * Inputs:
 *       none
 *
 * Outputs:
 *       none
 *
 * Callable From:     any task
 *
 * Reentrant:   No
 *
*=============================================================================
 ========
 */
PRIVATE void initLZWK(void)
{
  register int index;
  int i;
  register CodeDefT * CodeDefP;


  // empty the code definition tables
  for (index = 0; index < MAX_CODES; index++) {
       CodeDefP = &(CodeDefTbl[index]);
       CodeDefP->myCode   = index;
       CodeDefP->prefixCode  = NULL_CODE;
       CodeDefP->appendCode  = NULL_CODE;
       CodeDefP->siblingCode = NULL_CODE;
       CodeDefP->codeSize = 0;
       CodeDefP->codeStrP = (U8T*) CodeDefP;
       for (i = 0; i < 16; i++)
          CodeDefP->childCode[i] = NULL_CODE;
  }

  // Add the first 256 byte codes to the dictionary
```

```
        for (index = 0; index <= 255; index++) {
            addCodeLZWK(index, (U8T*) &(byteVals[index]), 1, NULL_CODE, index);
        }

        // The NULL_CODE definition is already initialized

        // for MIB, add these special reserved code to the dictionary:
        // ranging 2 .. (NUM_ZERO_CODES+1) zeros
        addCodeLZWK(257, (U8T *)zeroBytes, 2, 0, 0); // code 257: 2 zeros
        for (index = 258; index < FIRST_USER_CODE; index++) {
            addCodeLZWK(index, (U8T *) zeroBytes, index-255, index-1, 0);
        }

    }
```

```
/*
*=============================================================================
 5 ============
 * Function Name:     CompressLZWK
 *
 * Description:
 *       The LZWK compression (see notes at the top of this file)
10 *
 * Inputs:
 *       fInArray:       beginning address of the input string
 *       fOutArray       beginning address of the compressed buffer
 *       fInByteCount  size of input
15 *
 * Outputs:
 *       The byte count of the final compressed data
 *
 * Callable From:       any Task
20 *
 * Reentrant:   No
 *

*=============================================================================
25 ============
 */
U32T CompressLZWK(
  U8T * fInArray,     /* pointer to the input  byte array */
  U8T * fOutArray,    /* pointer to the output byte array */
30  U32T  fInByteCount   /* number of bytes of the uncompressed data */
  )
{
  U32T      bitsPerCode; /* variable code size */
  U32T      maxCode;
35  U16T      nextCode = FIRST_USER_CODE; // next unused code in dictionary
  U16T      code1 = NULL_CODE;
  U16T      code2 = NULL_CODE;
  U8T       * strP1 = fInArray;
  U8T       * strP2;
40  U32T      codeSize1 = 0;
  U32T      codeSize2 = 0;
  int       inByteCount = fInByteCount;
  U8T       * outArray = fOutArray;
  U32T      nextBitPos = 0; // 0 => LSb, 7 => MSb
45
```

```
    // initialize the dictionary
    initLZWK();

    // initialize bitsPerCode
    bitsPerCode = 9;
    maxCode = 1 << bitsPerCode;

    // read the first one-byte code, code1
    if (inByteCount > 0) {
        code1 = strP1[0];
        codeSize1 = 1;
    }

    while (code1 != NULL_CODE) {
        // Among the descendants of code1,
        // see if we can find a better match of code1
        code1 = FindBestFamilyMember(code1, strP1, inByteCount);
        codeSize1 = CodeDefTbl[code1].codeSize;
        inByteCount -= codeSize1;

        // output code1
        Send(code1, bitsPerCode, &outArray, &nextBitPos);

        if (inByteCount > 0) { /* there is more input */
          // now, code1 holds the first best-match string code
          // within the remaining input, get the first byte, and use it to
          // find the next best (longest) match string code, code2
          strP2 = &(strP1[codeSize1]);
          code2 = (U16T) (strP2[0]);
          code2 = FindBestFamilyMember(code2, strP2, inByteCount);
          codeSize2 = CodeDefTbl[code2].codeSize;

          // now we have best-matched code1 and code2.  If dictionary is
          // not full yet, add the string defined by code1::code2 as a
          // new code to the dictionary
          if (nextCode <= LAST_CODE) {
              addCodeLZWK(nextCode++, strP1,codeSize1+codeSize2,code1,code2);
              /* increment bitsPerCode if necessary */
              if ((nextCode > maxCode) && (bitsPerCode <
MAX_BITS_PER_CODE)){
                  bitsPerCode++;
                  maxCode = 1 << bitsPerCode;
              }
          }
```

```
            // get ready for the next iteration
            strP1 = strP2;
            code1 = code2;
            codeSize1 = codeSize2;
        } else { /* no more input */
            code1 = NULL_CODE;
        }
    }

    // output the EOF code
    Send(NULL_CODE, bitsPerCode, &outArray, &nextBitPos);

    if (nextBitPos == 0)
        return (outArray - fOutArray);
    else
        return (outArray - fOutArray + 1);
}
```

```
/*
*=============================================================================
=========
 * Function Name:     DecompressLZWK
 *
 * Description:
 *       The LZWK decompression (see notes at the top of this file)
 *
 * Inputs:
 *       fInArray:       beginning address of the input string
 *       fOutArray       beginning address of the compressed buffer
 *
 * Outputs:
 *       The byte count of the final decompressed data
 *
 * Callable From:      any Task
 *
 * Reentrant:   No
 *
*=============================================================================
=========
 */
U32T DecompressLZWK(
   U8T * fInArray,     /* pointer to the input  byte array */
   U8T * fOutArray     /* pointer to the output byte array */
   )
{

   U32T  bitsPerCode; /* variable code size */
   U32T  maxCode;
   U16T  code1 = NULL_CODE;
   U16T  code2 = NULL_CODE;
   U32T  codeSize1 = 0;
   U32T  codeSize2 = 0;
   U8T   * strP1;
   U8T   * strP2;
   U16T  nextCode = FIRST_USER_CODE; // the next unused code in the dictionary
   U32T  nextBitPos = 0; // 0 => LSb, 7 => MSb
   U8T   * inArray = fInArray;


   // initialize the dictionary
```

```
    initLZWK();

    // initialize bitsPerCode
    bitsPerCode = 9;
    maxCode = 1 << bitsPerCode;

    // read the first code, code1;
    code1 = Receive(bitsPerCode, &inArray, &nextBitPos);
    strP1 = fOutArray;
    codeSize1 = CodeDefTbl[code1].codeSize;
    if ( (code1 != NULL_CODE) && (code1 < nextCode) ) {
        // output the string represented by code1
        memcpy (strP1, CodeDefTbl[code1].codeStrP, codeSize1);

        // then read in the second code code2
        code2 = Receive(bitsPerCode, &inArray, &nextBitPos);
    } else { // error case handling
        return 0;
    }

    while (code2 != NULL_CODE) {
        strP2 = &(strP1[codeSize1]);

        if (code2 < nextCode) {
          // Normal case: output the string represented by code2
          codeSize2 = CodeDefTbl[code2].codeSize;
          memcpy (strP2, CodeDefTbl[code2].codeStrP, codeSize2);

          // now we have found code1::code2 in the input
          // add it to the dictionary
          if (nextCode <= LAST_CODE) {
              addCodeLZWK(nextCode++, strP1,codeSize1+codeSize2,code1,code2);
              /* adjust bitsPerCode if necessary */
              if ((nextCode >= maxCode) && (bitsPerCode
<MAX_BITS_PER_CODE)){
                  bitsPerCode++;
                  maxCode = 1 << bitsPerCode;
              }
          }

          // get ready for the next iteration
          strP1 = strP2;
          code1 = code2;
          codeSize1 = codeSize2;
          code2 = Receive(bitsPerCode, &inArray, &nextBitPos);
```

```
        } else if (code2 == nextCode) {
          // special case handling: if we are using the code that just
          // got generated at the data compressor, but not yet
          // generated on the decompression side yet, this code
          // code2 must represent code1::code1, add code2 to dictionary
          memcpy (strP2, strP1, codeSize1);
          memcpy (&(strP2[codeSize1]), strP1, codeSize1);
          codeSize2 = codeSize1 * 2;
          addCodeLZWK(nextCode++, strP1, codeSize2, code1, code1);
          /* adjust bitsPerCode if necessary */
          if ((nextCode >= maxCode) && (bitsPerCode < MAX_BITS_PER_CODE)){
              bitsPerCode++;
              maxCode = 1 << bitsPerCode;
          }

          // get ready for the next iteration
          strP1 = strP2;
          code1 = code2;
          codeSize1 = codeSize2;
          code2 = Receive(bitsPerCode, &inArray, &nextBitPos);
        } else { // error case handling: undefined code
          return 0;
        }
    }

    // output the size of the decompressed data
    return (strP1 + codeSize1 - fOutArray);

}
```